GENERAL

core java interview questions for 3 years experience

Published: September 3, 2026 | Index ID: #-

Core Java Interview Questions for 3 Years Experience

Landing a Java developer role after three years of professional experience can feel like a balancing act. On one hand, you've moved beyond the beginner's learning curve and have hands on experience building real world applications. On the other hand, interviewers often expect you to demonstrate a deeper understanding of core Java concepts, design patterns, and best practices. This guide compiles the most frequently asked interview questions for developers with around three years of experience, along with clear explanations and practical examples. Whether you're preparing for a technical interview, a coding challenge, or a behavioral round that dives into your Java knowledge, this article will help you articulate your expertise confidently.

Why Focus on Core Java?

While many companies now use Spring, Hibernate, or micro service architectures, the foundation of every Java application is still the Java Development Kit (JDK). Core Java covers:

- Object oriented principles
- Collections framework
- Concurrency and synchronization
- Memory management and garbage collection
- Exception handling
- Java 8+ features (lambdas, streams, Optional)
- Design patterns and architectural considerations

Mastering these topics demonstrates that you can write clean, maintainable code and troubleshoot performance or scalability issues—skills that interviewers value highly.

How to Use This Guide

1. Read through the sections that align with your current knowledge gaps.
2. Practice coding the examples in an IDE or an online compiler.
3. Rehearse explaining the concepts aloud—this will help during live interviews.
4. Keep the **"Key Takeaway"** bullet in each section handy for quick revision.

1. Object-Oriented Programming (OOP) Fundamentals

1.1. What is Polymorphism and How Does It Work in Java?

Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. In Java, it manifests through method overriding (runtime polymorphism) and method overloading (compile time polymorphism).

1. Method Overriding: A subclass provides its own implementation of a method defined in its superclass. The JVM resolves the method at runtime based on the actual object''s type.
2. Method Overloading: Multiple methods with the same name but different parameter lists within the same class. The compiler chooses the appropriate method at compile time.

Key Takeaway: Polymorphism promotes code reuse and flexibility. In interviews, be ready to show how you've

used it to implement strategy patterns or to create flexible APIs.

1.2. Explain the Difference Between Abstract Class and Interface.

Both are used to achieve abstraction, but they differ in several ways:

- Abstract Class

  Can contain both abstract and concrete methods.
- Can have instance variables with any visibility.
- Supports constructors.
- Single inheritance only.

- Before Java 8: only abstract methods (no implementation).
- Java 8+: default and static methods allowed.
- All methods are implicitly public.
- Supports multiple inheritance of type.

Use an interface when you need to define a contract that multiple unrelated classes can implement. Use an abstract class when you want to share common code or state.

Key Takeaway: Clarify when you'd choose one over the other, especially in design decisions like defining a service layer.

1.3. What Is the Difference Between Composition and Aggregation?

Both are ways to model relationships between objects.

- Aggregation (has a relationship) represents a weak association. The lifetime of the contained object is independent of the container. Example: A Library aggregates Book objects.
- Composition (part of relationship) is a strong association where the contained object cannot exist without the container. Example: A Car composes an Engine; if the car is destroyed, the engine is also gone.

In interviews, you might be asked to choose between them when modeling domain entities.

Key Takeaway: Composition enforces strong ownership; aggregation indicates shared ownership.

1.4. Explain the SOLID Principles.

SOLID is a set of five design principles that promote maintainable and scalable software.

1. Single Responsibility Principle (SRP): A class should have only one reason to change.
2. Open/Closed Principle (OCP): Classes should be open for extension but closed for modification.
3. Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types.
4. Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use.
5. Dependency Inversion Principle (DIP): High level modules should not depend on low level modules; both should depend on abstractions.

When answering, provide a real world example—perhaps refactoring a service layer to meet SRP or using interfaces to satisfy DIP.

Key Takeaway: SOLID principles guide you to write clean, loosely coupled code.

2. Collections Framework

2.1. What Are the Differences Between ArrayList, LinkedList, and Vector?

All three implement the List interface but differ in performance characteristics:

- ArrayList

  Backed by a resizable array.
- Fast random access (O(1)).
- Insertion/deletion at the end is amortized O(1); at arbitrary positions is O(n).
- Not synchronized.

- Double linked list implementation.
- Insertion/deletion at both ends is O(1).
- Random access is O(n).
- Not synchronized.

- Similar to ArrayList but synchronized.
- Thread safe but slower due to synchronization overhead.
- Legacy class—use Collections.synchronizedList(new ArrayList()) instead.

Choose based on your use case: ArrayList for most scenarios, LinkedList for frequent insertions/deletions in the middle, and Vector only if legacy code demands it.

Key Takeaway: Understanding underlying data structures helps you pick the right list for performance.

2.2. Explain the Difference Between HashMap, LinkedHashMap, and TreeMap.

All three implement the Map interface but differ in ordering and performance.

- HashMap

  Unordered; provides O(1) average time for get/put.
- Uses hash codes; collisions handled via chaining.

- Maintains insertion order (or access order if configured).
- Same performance as HashMap.

- Sorted according to natural ordering of keys or a provided Comparator.
- Operations are O(log n) due to Red Black tree implementation.

When interviewers ask, be ready to explain a scenario where you'd need sorted keys (TreeMap) or predictable iteration order (LinkedHashMap).

Key Takeaway: Map choice depends on key ordering and performance requirements.

2.3. What Is the Difference Between Iterator and ListIterator?

Both provide traversal capabilities but differ in scope and functionality.

- Iterator

  Can traverse forward only.
- Provides hasNext(), next(), and remove().

- Specific to List implementations.
- Bidirectional traversal: hasPrevious(), previous().
- Can modify the list during iteration: add(), set().

Use ListIterator when you need to navigate backward or modify elements while iterating.

Key Takeaway: Understand the capabilities of each iterator to avoid ConcurrentModificationException.

3. Concurrency and Multithreading

3.1. What Is the Difference Between Thread, Runnable, and Callable?

All three represent units of work but differ in execution and result handling.

- Thread

  Encapsulates a Runnable or extends itself.
- Has lifecycle methods: start(), join(), sleep().
- Does not return a value.

- Functional interface with a single run() method.
- Can be passed to a Thread or an ExecutorService.
- No return value.

- Functional interface with call() that returns a value and may throw exceptions.
- Used with ExecutorService to obtain a Future.

In interviews, you might be asked to explain why you'd use Callable for tasks that produce a result.

Key Takeaway: Thread = execution context; Runnable = task without return; Callable = task with return.

3.2. Explain the Java Memory Model and Happens Before Relationship.

The Java Memory Model (JMM) defines how threads interact with memory, ensuring visibility and ordering of operations.

- Visibility: Changes made by one thread to shared variables become visible to others.
- Ordering: Ensures that actions occur in a predictable order.

The **happens before** relationship is a partial ordering that guarantees visibility. For example:

1. Actions in a thread before a synchronized block happen before actions after the block in another thread.

Baca Juga (Artikel Terkait)

- **automaker mini software version v8 erptoboss**
  URL: http://art-consciousness.com/automaker-mini-software-version-v8-erptoboss.pdf
- **automata and computability**
  URL: http://art-consciousness.com/automata-and-computability.pdf
- **automata and languages theory and applications**
  URL: http://art-consciousness.com/automata-and-languages-theory-and-applications.pdf
- **automata and mechanical toys**
  URL: http://art-consciousness.com/automata-and-mechanical-toys.pdf

Tags: #core #java #interview #questions #years #experience